A Compiler Algorithm for Managing Asynchronous Memory Read Completion[1]

Daniel Maskit
Scalable Concurrent Programming Laboratory
California Institute of Technology

January 15, 1996

Abstract

Computers with conventional memory systems have a predictable latency between initiation and completion of a memory read. On such machines it is relatively easy for either the compiler or the processor to guarantee that a load has completed before further references to the loaded register are made. In a machine with a logically shared, but physically distributed, memory, these latencies are not statically predictable. Some existing systems, such as the Cray T3D, deal with this problem by using a hardware mechanism to enforce synchronization on a register which is the target of a remote memory access. The M-Machine currently being designed by the Concurrent VLSI Architecture Group at MIT performs remote memory accesses asynchronously, and allows program execution to continue while the access is outstanding, but does not enforce synchronization in hardware. This architectural simplification, and resulting relaxation of memory completion semantics, poses a challenge to the compiler: how can this simpler memory system be efficiently supported while maintaining program correctness. In particular, what is required to guarantee that there are no conflicts between completion of a memory operation by placing a value into a register, and other uses of the register being written. This paper describes a general solution to this problem, develops an algorithm to implement it, and shows that the algorithm is correct.

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency under contract number DABT63-95-C-0116. The information contained herein does not necessarily reflect the position or policy of the government of the United States, and no official endorsement should be inferred.

1 Overview

Computers with conventional memory systems have a predictable latency between initiation and completion of a memory read. On such machines it is relatively easy for either the compiler or the processor to guarantee that a load has completed before further references to the loaded register are made. In a machine with a logically shared, but physically distributed memory, these latencies are not statically predictable. The standard method for dealing with this issue in hardware is to have operations check the synchronization state of their destination register and block until the register is in a stable state. This technique is used on machines such as the Cray T3D.

The M-Machine currently being designed by the Concurrent VLSI Architecture Group at MIT will perform remote memory accesses asynchronously, but does not check the synchronization state of registers prior to overwriting them. This allows processing to continue while awaiting completion of a remote memory access, but does not require the hardware complexity to allow reading the synchronization state prior to writing over the register. This architectural decision is necessitated by the use of multiple loosely-coupled processor clusters on a single chip. The necessary hardware interlocks to manage memory synchronization could not be implemented without significant complications. This architectural simplification, and resultant relaxation of memory completion semantics, poses a challenge to the compiler: how can this memory system be efficiently supported while maintaining program correctness? In particular, what is required to guarantee that there are no conflicts between completion of a memory operation and other uses of the destination for that operation?

This paper describes a general solution to this problem, develops an algorithm to implement it, and shows that the algorithm is correct.

2 Problem description

This section describes the problem of write-after-write hazards. As part of this description some terminology is introduced. This terminology is used for formulating the solution to the problem.

The following definitions will facilitate discussion of this issue. The set of possible machine operations can be divided into three categories that differ in their handling of synchronization state of registers. Within this paper, `READ` operations are defined to be the only operations that perform synchronization.

- `LOAD` places a value into a *destination* register. The completion of a `LOAD` can occur significantly after its initiation. When a `LOAD` is initiated it sets a synchronization flag associated with the *destination* register. This flag is cleared when the `LOAD` completes.

- `READ` takes a value from a *source* register and uses it to perform some operation (such as *add, subtract, compare, etc.*). `READS` will not begin until the synchronization flag on the *source* register is cleared.
- `WRITE` places a value into a *destination* register. `WRITES` will complete even if the synchronization flag on the *destination* register was set when they started. After a `WRITE` has completed, the synchronization flag on the *destination* register is always cleared.

2.1 Problem 1: Delayed Loads

Figure 1 shows a code fragment which illustrates a problem with this system: if a `WRITE` has as its *destination* a register which has its synchronization flag set, this register could end up in an undesired state. The value **p* is `LOADED` into *r1*. If $q = 0$, the program will stall until the `LOAD` completes, and then `READ` *r1* and add it to *r2*. If $q \neq 0$, *r1* could be `WRITTEN` before the `LOAD` completes. Figure 2 shows a worst-case timing sequence. In this case, in between the literal *#1* being `WRITTEN` to *r1*, and *r1* being returned, the `LOAD` completes, overwriting the value *#1* in *r1*. The result is that even though $q \neq 0$, the return value is **p*, rather than the expected #1.

```
a = *p;              LOAD      p, r1          ; LOAD from location p into r1
if(q == 0)           EQUAL     q, 0, cc0      ; Check (q == 0)
                     BF        cc0, L1        ; If false, branch to L1
    a = a + b;       ADD       r2, r1, r2     ;Add r1 to r2
                     JMP       L2
else                 L1:
    a = 1;           MOVE      #1, r1         ; WRITE r1
                     L2:
return a;            RETURN    r1
```

Figure 1: Delayed Load

2.2 Problem 2: Multiple Loads

Figure 3 shows a second problem: If a `LOAD` to a register is followed by a second `LOAD` to the same register, the contents of the register will be unknown. Here, the value **p* is `LOADED` into *r1*. If $q = 0$, the value *b* could be `LOADED` into *r1* before the load of **p* completes. In this case, in between *b* being `LOADED` into *r1*, and *r1* being `READ`, the `LOAD` of **p* completes, overwriting the value in *r1*. The result is that even though $q = 0$, the return value is $r4 + *p$, rather than the expected $r4 + *b$.

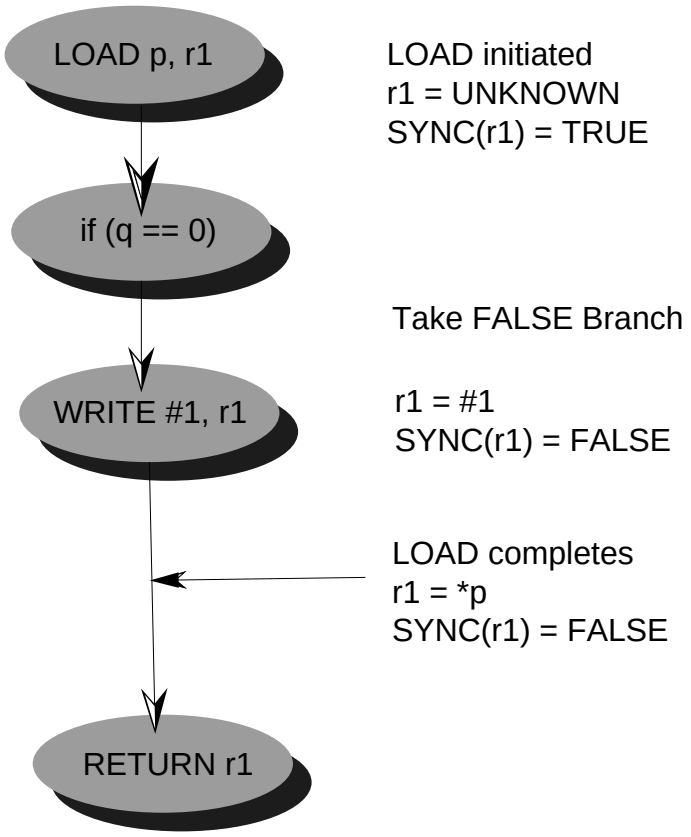

Figure 2: Worst-case Timing for Delayed Load

```
a = *p;          LOAD     p, r1        ; LOAD from location p into r1
if(q == 0)       EQUAL    q, 0, cc0    ; Check (q == 0)
                 BF       cc0, L1      ; If false, branch to L1
    a = *b;      LOAD     b,r1
                 L1:
a = a + c;       ADD      r1,r4,r4     ; READ r1, r4; WRITE r4
return a;        RETURN   r4
```

Figure 3: Multiple Loads

The condition that needs to be met can be stated as:

> Guarantee that no `WRITE` or `LOAD` has as its *destination* a register whose synchronization flag is set.

This can be achieved within a basic block using a single forward pass over the block, and inserting correction code when a problematic instruction is encountered. To achieve correctness `READS` can be inserted to force synchronization. However, complications arise when dealing with transitions between basic blocks, as this requires transmitting state information across block boundaries.

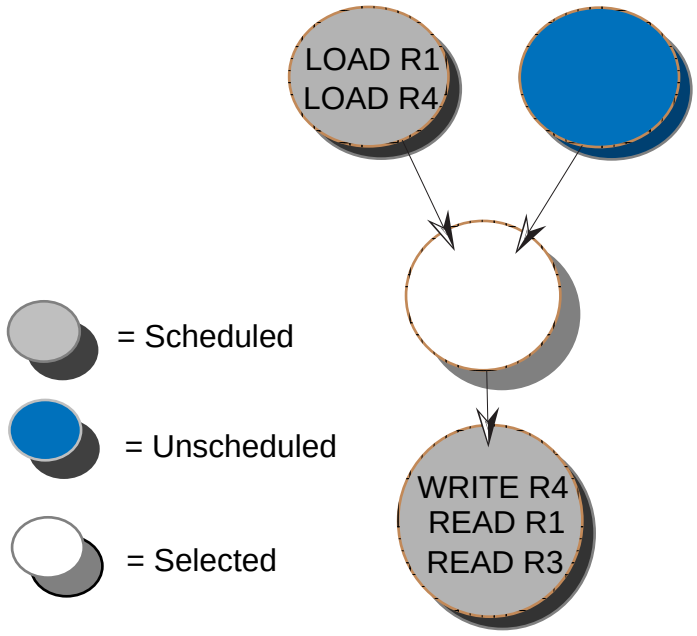

Figure 4: Adjacent Block Scheduling

In addition, transitions between blocks must also be managed. To provide for a more general treatment, the scheduling of basic blocks is assumed to have no fixed ordering. Therefore, any or all of these other basic blocks might already be scheduled. This situation is illustrated in Figure 4. This figure shows four basic blocks. Two of these basic blocks have already been scheduled; two of the blocks are as yet unscheduled. One of the unscheduled blocks has been selected as the next block to be scheduled. As can be seen, there are operations in both the preceding and succeeding scheduled block which might require action within the current block. For example, if the first instruction in the current block is a `WRITE R4`, this is one of the situations which requires an inserted `READ`. Similarly, if the last instruction is a `LOAD R3`, the state of the predecessor guarantees that this is a safe operation to perform.

The problem thus becomes:

Ensure that register states from scheduled predecessors are respected. Generate a schedule for the current block, inserting synchronizing `READS` where required. Examine scheduled successors, insert any required `READS` to guarantee proper transition from current block into scheduled blocks. Ensure that the proper information is made available to all predecessors and successors that have not yet been scheduled.

3 Overview of solution

The solution to this problem can be broken into three tasks of definition: define how to schedule the code for a basic block; define what information needs to be transmitted from one block to another; and define how to handle information from a neighboring block which is already scheduled. The key to resolving the problematic state transitions is to insert code to perform a `READ` so as to guarantee synchronization.

There are three possible states for a given register. Each of these states can be determined by being inherited from a predecessor, established in the current block, or upwardly-exposed from a successor.

- `HOT`. The synchronization bit for the register is set; or the upwardly-exposed reference to this register is a `LOAD`.
- `COLD`. The synchronization bit for the register is cleared. There either is no upwardly exposed reference, or the exposed reference is a `WRITE`.
- `GROUNDED`. The exposed reference to this register is a `READ`.

In general, the final state of registers for a basic block is determined by the initial state of the registers inherited from any predecessors that have already been scheduled, and the set of operations that is performed on each register within the block. Once this state has been determined, there is one possible complication that can arise: if one or more successors have been scheduled, a check must be made to ensure that there are no conflicts between the final state for the current block and the initial state of the scheduled successors. This is done by examining the *upwardly-exposed* state of each register in the successors. This state is determined by the first use of each register within the successor.

While within a block, it is necessary to have available information about the block's successors and predecessors. It is possible to structure a solution so that only immediately adjoining blocks which have been scheduled are of interest. If an adjoining block has been scheduled, the exposed state within the scheduled block of all registers is relevant.

There are two types of information that need to be communicated between adjoining blocks. The first type of information is a list of registers which are `HOT` on exit from the block. This information is called $\texttt{HOT}_\texttt{b}$, and can flow from a predecessor into the current block, or from the current block into a successor. The second type of information is what

type of operation is performed as the first access to each register within a given block. This information is called $\texttt{STATE}_\texttt{b}$, and can flow from the current block to a predecessor, or from a successor to the current block.

If a predecessor has not been scheduled, there is no initial flow of information between the predecessor and the current block. When scheduling is completed, $\texttt{STATE}_\texttt{b}$ is made available to the predecessor.

If a successor has not been scheduled, there is no initial flow of information between the successor and the current block. When scheduling is completed, $\texttt{HOT}_\texttt{b}$ is made available to the successor.

If a predecessor has been scheduled, $\texttt{HOT}_\texttt{p}$ flows from the predecessor to the current block. The initial value for $\texttt{HOT}_\texttt{b}$ for the current block is the union of $\texttt{HOT}_\texttt{p}$ from all scheduled predecessors.

If a successor has been scheduled, $\texttt{STATE}_\texttt{s}$ flows from the successor to the current block.

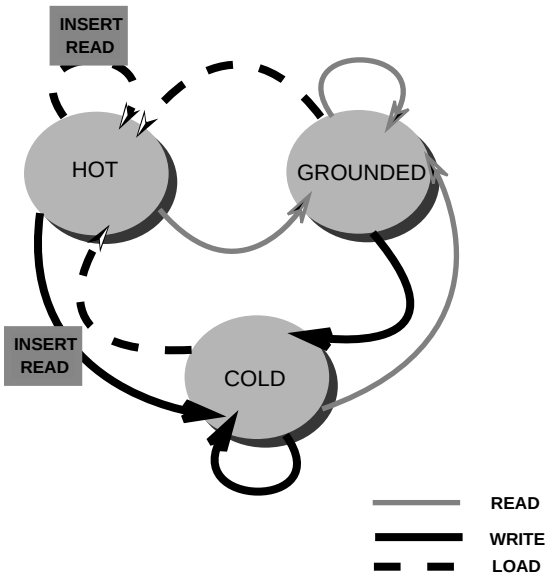

Figure 5: State Transition Diagram

If a successor has already been scheduled, a determination needs to be made as to whether the ending state from the current block is consistent with the first usage in the next block. Leaving the current block, each register is identified as either `HOT` or `COLD`. If a register is `COLD` on exit from the current block, than any initial operation using that register in the next block is correct. If a register is `HOT`, however, the only initial operation that is legal is a `READ`. If the upwards-exposed reference to a register is a `READ`, the register state is `GROUNDED`. If the `STATE` for a register in all scheduled successors is `GROUNDED`, than it is acceptable for the register to be `HOT` on exit from the current block. Otherwise, it is necessary to `GROUND` the register by performing a `READ` on the register prior to exiting the current block. The full set of state transitions is shown in Figure 5.

4 Algorithm

Recall that the problem being solved is:

> Ensure that register states from scheduled predecessors are respected. Generate a schedule for the current block, inserting synchronizing `READS` where required. Examine scheduled successors, insert any required `READS` to guarantee proper transition from current block into scheduled blocks. Ensure that the proper information is made available to all predecessors and successors that have not yet been scheduled.

To facilitate formulation of an algorithm to solve this problem, registers are defined as being in one of three states: `HOT` registers have their synchronization bit set; `GROUNDED` registers had their synchronization bit cleared by a `READ`; `COLD` registers had their synchronization bit cleared by a `WRITE`. The state `GROUNDED` is only relevant for managing transitions from a block to a scheduled successor; and only needs to appear in the `STATE` passed from a scheduled successor to the current block. Within a block, it is sufficient to know if a register is `HOT` or `COLD`.

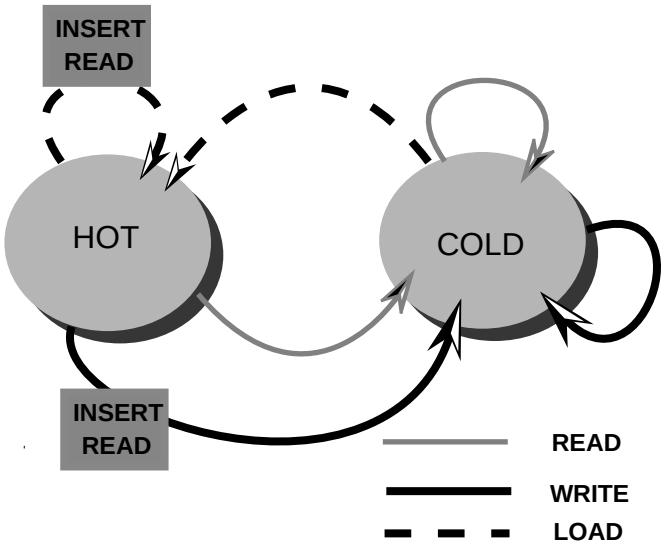

Figure 6: State Transitions for Single-Instruction Scheduling

At any point in the scheduling of a block, there will be some set of instructions which have already been scheduled, a set $\texttt{HOT\_current}_b$ of registers with their synchronization bit set, and a next instruction to be scheduled. The possible state transitions for each register referenced in the next instruction are shown in Figure 6. For any given register, if the register is not in $\texttt{HOT\_current}_b$ initially, any operation can be performed without special handling. If the performed operation is a `LOAD`, the register is placed in $\texttt{HOT\_current}_b$. If the register is initially in $\texttt{HOT\_current}_b$, the only operation which can be performed

without special handling is a synchronizing READ. Any other operation will destroy the synchronization state of the register. To compensate for this, a READ must be inserted prior to issuing the instruction so that the register synchronization state is cleared before the register is overwritten. If either a READ or a WRITE is performed, the register is removed from the set $\texttt{HOT\_current}_b$.

The general outline of the algorithm is shown in Figure 7. Iterating until all basic blocks have been scheduled, select a basic block. The routine SelectBlock() chooses the next block to be scheduled. The criteria used to make this selection need not be specified as they do not affect the workings of the algorithm. It is guaranteed that the return value from SelectBlock() is a block which has not yet been scheduled. Calculate $\texttt{HOT\_initial}_b$ for the selected block b. Schedule the instructions within this basic block. Calculate $\texttt{STATE\_incoming}_b$ for this block. For any register which was not referenced in this block, set $\texttt{STATE}_b(r)$ for that register to $\texttt{STATE\_incoming}_b(r)$. Generate any compensation code needed to reconcile $\texttt{HOT\_current}_b$ with $\texttt{STATE\_incoming}_b$.

```
while (not all blocks scheduled)
    b = SelectBlock();
    HOT_initial_b = ∅;
    for all predecessors p of b
        if (predecessor p scheduled)
            HOT_initial_b = HOT_initial_b ∪ HOT_finalp;
    HOT_current_b = HOT_initial_b

    Schedule(b);
    STATE_incoming_b = JoinSuccessorStates(b);

    for all registers r
        if (r ∈ HOT_current_b AND (STATE_incoming_b(r) ≠ GROUNDED)
            issue read of r
            remove r from HOT_current_b
            if (STATE_b(r) = NULL)
                STATE_b(r) = GROUNDED;
        if (STATE_b(r) = NULL)
            STATE_b(r) = STATE_incoming_b(r);
    HOT_final_b = HOT
    Mark b as scheduled
```

Figure 7: General Outline of Algorithm

The algorithm for scheduling the instructions within a basic block is shown in Figure 8. This algorithm iterates over all of the instructions within a basic block, ensuring that all operations are scheduled and all necessary READS are inserted. In addition, this algorithm determines the value of $\texttt{STATE}_b(\texttt{r})$ for all registers referenced in the block, and maintains the membership information for the set $\texttt{HOT\_current}_b$.

```
for each operation O {
    if O is a READ {
        if STATE_b(source(O)) = NULL
            STATE_b(source(O)) = GROUNDED
        if STATE_b(target(O)) = NULL
            STATE_b(target(O)) = COLD
        if source(O) ∈ HOT_current_b
            remove source(O) from HOT_current_b
    }

    if O is a WRITE {
        if target(O) ∈ HOT_current_b {
            issue READ of target(O)
            remove target(O) from HOT_current_b
        }
        if STATE_b(target(O)) = NULL
            STATE_b(target(O)) = COLD
    }

    if O is a LOAD {
        if target(O) ∈ HOT_current_b {
            issue READ of target(O)
            if STATE_b(target(O)) = NULL
                STATE_b(target(O)) = GROUNDED
        }
        else {
            add target(O) to HOT_current_b
            if STATE_b(target(O)) = NULL
                STATE_b(target(O)) = HOT
        }
    }

    issue O
}
```

Figure 8: Scheduling Algorithm for a Basic Block

The routine `JoinSuccessorStates()` is shown in Figure 9. This routine examines `STATE` for all scheduled successors, and merges them into one `STATE`.

```
for all registers r
    STATE_b(r) = GROUNDED;
for all scheduled successors s of b
    for all registers r
        if (STATE_b(r) = HOT ∨ STATE_s(r) = HOT)
            STATE_b(r) = HOT;
        else if (STATE_b(r) == COLD ∨ STATE_s(r) == COLD)
            STATE_b(r) = COLD;
return STATE_b;
```

Figure 9: Algorithm for Merging STATE from Scheduled Successors

5 Correctness of the Algorithm

This section demonstrates that the algorithm presented in the previous section will guarantee that the synchronization state of a register is always cleared before an operation using that register as its *destination* is issued. This is shown by providing more formal definitions for the concepts discussed in previous sections, describing invariants for the algorithm, and then providing a demonstration of correctness.

5.1 Basic Definitions

This following provides more formal definitions for many of the concepts introduced in the previous several sections.

B is the complete set of basic blocks in a program.

R is the complete set of available machine registers.

SCHEDULED(b) = TRUE if code has been generated for block b.

FIRST(b, r), $b \in \mathbf{B}, r \in \mathbf{R}$, is the first operation in a block b that references r as *source* and/or *destination*. FIRST(b, r) is one of `READ, WRITE, LOAD`.

LAST(b, r), $b \in \mathbf{B}, r \in \mathbf{R}$, is the last operation in a block b that references r as *source* and/or *destination*. LAST(b, r) is one of `READ, WRITE, LOAD`.

SOURCE(OP(b, r)) = TRUE if r is a *source* for OP(r) in b.
OP $\in$ {FIRST,LAST}, $r \in \mathbf{R}, b \in \mathbf{B}$.

DESTINATION(OP(b,r)) = TRUE if r is a *destination* for OP(r) in b.
OP $\in$ {FIRST,LAST}, $r \in \mathbf{R}, b \in \mathbf{B}$.

SUCCESSOR(b,s) = TRUE if $b,s \in \mathbf{B} \wedge s$ is a successor of b.

PREDECESSOR(b,p) = TRUE if $b,p \in \mathbf{B} \wedge p$ is a predecessor of b.

If an instruction has a register as both *source* and *destination*, the state of the register after the instruction has been performed is determined as if the register were only the *destination*. The use of the register as *source* ensures that correction code will not be necessary prior to executing the instruction.

Initially $\texttt{STATE}_\texttt{b} = \emptyset$

5.2 Definitions for Transitional Information

Given these definitions, it is possible to define rules for determining the state visible from outside a scheduled block. First, the definition of $\texttt{STATE}_\texttt{b}$, which is the upwardly exposed register information for the block b. If the first reference to a register performs a `READ` on the register, then $\texttt{STATE}_\texttt{b}(r)$ for that register is `GROUNDED`:

SOURCE(FIRST(b,r)) = TRUE $\rightarrow \texttt{STATE}_\texttt{b}(r) = \texttt{GROUNDED}$

If the first instruction that accesses a register uses that register as both *source* and *destination*, $\texttt{STATE}_\texttt{b}(r)$ is `GROUNDED`, as the subsequent use of the register as *destination* is not the first reference to the register. If the first reference to a register is as *destination*, the operation could be either a `WRITE` or a `LOAD`. If it is a `WRITE`, then $\texttt{STATE}_\texttt{b}(r)$ is `COLD`, if it is a `LOAD`, then $\texttt{STATE}_\texttt{b}(r)$ is `HOT`:

SOURCE(FIRST(b,r)) = FALSE $\wedge$ FIRST$(b,r) = \texttt{WRITE} \rightarrow \texttt{STATE}_\texttt{b}(r) = \texttt{COLD}$

SOURCE(FIRST(b,r)) = FALSE $\wedge$ FIRST$(b,r) = \texttt{LOAD} \rightarrow \texttt{STATE}_\texttt{b}(r) = \texttt{HOT}$

Once it is possible to derive the definition of $\texttt{STATE}_\texttt{b}$ for one block, it is possible to combine the $\texttt{STATE}_\texttt{s}$ for all scheduled successors of that block once some rules of precedence are defined. If $\texttt{STATE}_\texttt{p}(r) = \texttt{HOT}$ for any predecessor, $\texttt{STATE\_incoming}_\texttt{b}(r) = \texttt{HOT}$. If $\texttt{STATE}_\texttt{p}(r) \neq \texttt{HOT}$ for all predecessors, but $\texttt{STATE}_\texttt{p}(r) = \texttt{COLD}$ for any predecessor, $\texttt{STATE\_incoming}_\texttt{b}(r) = \texttt{COLD}$. Otherwise, $\texttt{STATE}_\texttt{p}(r) = \texttt{GROUNDED}$ for all predecessors, and $\texttt{STATE\_incoming}_\texttt{b}(r) = \texttt{GROUNDED}$. Given these rules, it is possible to define:

$$\begin{aligned}\texttt{STATE\_incoming}_\texttt{b} \;=\; & \{\cap_i \texttt{STATE}_{\texttt{b}_\texttt{i}} \mid b_i \in \mathbf{B} \wedge \text{SCHEDULED}(b_i) = \text{TRUE} \\ & \wedge \text{SUCCESSOR}(b, b_i) = \text{TRUE}\end{aligned}$$

There are two rules needed for determining $\texttt{HOT\_final}_\texttt{b}$. If the last reference to a register uses the register as *destination*, and the operation is a `LOAD`, then $\texttt{HOT\_final}_\texttt{b}(r)$ for that register is TRUE. In all other cases, $\texttt{HOT\_final}_\texttt{b}(r)$ for that register is FALSE:

$$\begin{aligned}
\text{DESTINATION}(\text{LAST}(b,r)) &= \text{TRUE} \wedge \text{LAST}(b,r) = \texttt{LOAD} \\
&\quad \rightarrow \texttt{HOT_final}_\text{b}(r) = \text{TRUE} \\
\text{DESTINATION}(\text{LAST}(b,r)) &= \text{FALSE} \vee \text{LAST}(b,r) \neq \texttt{LOAD} \\
&\quad \rightarrow \texttt{HOT_final}_\text{b}(r) = \text{FALSE}
\end{aligned}$$

For a given block, $\texttt{HOT\_initial}_\text{b}$ is the union of the sets $\texttt{HOT\_final}_\text{p}$ from all of the previously scheduled successors. For this union operation, a register is considered to be in $\texttt{HOT\_final}_\text{p}$, iff $\texttt{HOT\_final}_\text{p}(r) = \text{TRUE}$. This leads to the definition:

$$\begin{aligned}
\texttt{HOT_initial}_\text{b} &= \{\cup_i \texttt{HOT_final}_{\text{b}_\text{i}} \mid b_i \in \mathbf{B} \wedge \text{SCHEDULED}(b_i) = \text{TRUE} \\
&\quad \wedge \text{PREDECESSOR}(b, b_i) = \text{TRUE}\}
\end{aligned}$$

5.3 Invariants, Preconditions and Postcondition

The preconditions for the algorithm are:

- $\mathbf{B}$ contains all of the basic blocks for a procedure.
- The graph containing $\mathbf{B}$ is properly organized to represent the relationships between all blocks and their predecessors/successors
- Each $b \in \mathbf{B}$ contains a set of operations $\{O | O \in \{\texttt{READ}, \texttt{WRITE}, \texttt{LOAD}\}\}$
- All operations are legal to issue
- $b \in \mathbf{B} \rightarrow \text{SCHEDULED}(b) = \text{FALSE}\}$

The invariants for this algorithm are:

- $b \in \mathbf{B} \wedge$ SCHEDULED$(b) =$ TRUE $\rightarrow$ All original operations in b have been issued.
- $b \in \mathbf{B} \wedge$ SCHEDULED$(b) =$ TRUE $\rightarrow$ All required READS have been inserted in b.
- $b \in \mathbf{B} \wedge$ SCHEDULED$(b) =$ TRUE $\rightarrow$ $\texttt{HOT\_final}_\text{b}$ is correct.
- $b \in \mathbf{B} \wedge$ SCHEDULED$(b) =$ TRUE $\rightarrow$ $\texttt{STATE}_\text{b}$ is correct.

As long as these invariants are preserved, there is only one necessary postcondition for the algorithm:

- $b \in \mathbf{B} \rightarrow \text{SCHEDULED}(b) = \text{TRUE}$

5.4 Demonstration of Correctness

Initially, no blocks have been scheduled, and all invariants are trivially preserved. If $\mathbf{B} = \emptyset$, then there are no basic blocks within the current procedure, and the algorithm trivially terminates. Otherwise, there is at least one block which needs to be scheduled and the outer loop denoted by:

while (not all blocks scheduled)

will be entered. This loop condition can be rewritten as:

while ($\{\exists b | b \in \mathbf{B} \wedge$ SCHEDULED$(b) =$ FALSE $\}$)

Since `SelectBlock()` is guaranteed to return a value $\{b | b \in \mathbf{B} \wedge$ SCHEDULED$(\mathbf{B}) =$ FALSE$\}$, and such a b exists, after execution of this line it can be asserted that: $b \in \mathbf{B} \wedge$ SCHEDULED$(\mathbf{b}) =$ FALSE.

The next step in the algorithm is to ensure that the set $\texttt{HOT\_initial}_\texttt{b}$ contains only registers that are in $\texttt{HOT\_final}_\texttt{p}$ of scheduled predecessors p, and contains all such registers. The initialization:

$\texttt{HOT\_initial}_\texttt{b} = \emptyset$;

allows us to assert: $\texttt{HOT\_initial}_\texttt{b}$ contains no registers not present in $\texttt{HOT\_final}_\texttt{p}$ of a scheduled predecessor of b. If there are no scheduled predecessors of b, then this initial value is correct. If there are scheduled predecessors, the following code will ensure that $\texttt{HOT\_initial}_\texttt{b}$ contains the proper set of registers:

for all predecessors p of b
    if (predecessor scheduled)
        $\texttt{HOT\_initial}_\texttt{b} = \texttt{HOT\_initial}_\texttt{b} \cup \texttt{HOT\_final}_\texttt{p}$;

This can be expressed as:

($\{\forall p \mid p \in \mathbf{B} \wedge$ PREDECESSOR$(b, p) =$ TRUE $\}$)
    if (SCHEDULED$(p) =$ TRUE)
        $\texttt{HOT\_initial}_\texttt{b} = \texttt{HOT\_initial}_\texttt{b} \cup \texttt{HOT\_final}_\texttt{p}$;

This loop will terminate as it iterates only once for each predecessor of b, and the set of predecessors is finite. After execution of this loop the following can be asserted:

$$r \in \mathbf{R} \wedge r \in \texttt{HOT\_initial}_\texttt{b} \leftrightarrow r \in \texttt{HOT\_final}_\texttt{p} \wedge \text{SCHEDULED}(p) = \text{TRUE} \wedge \text{PREDECESSOR}(b, p) = \text{TRUE}$$

At this point, the computation of $\texttt{HOT\_initial}_\texttt{b}$ is complete. After the assignment:

$\texttt{HOT\_current}_\texttt{b} = \texttt{HOT\_initial}_\texttt{b}$

everything that has been asserted about $\texttt{HOT\_initial}_\texttt{b}$ will hold for $\texttt{HOT\_current}_\texttt{b}$ until the latter is modified.

It will be demonstrated in Section 5.4.1 that, after the call:

`Schedule(`b`);`

the following can be asserted: all original operations in b have been issued; all required `READS` have been inserted in b; $\texttt{HOT\_current}_\texttt{b}$ is correct up to this point; and $\texttt{STATE}_\texttt{b}$ is correct up to this point.

It will be shown in Section 5.4.2 that, following the call:

$\texttt{STATE\_incoming}_\texttt{b}$ `= JoinSuccessorStates(`b`);`

the following can be asserted: $\texttt{STATE\_incoming}_\texttt{b}$ is correct.

Next, an iteration over all registers is performed to identify any registers which are in $\texttt{HOT\_current}_\texttt{b}$, but are not grounded in $\texttt{STATE\_incoming}_\texttt{b}$. In addition, any registers for which $\texttt{STATE}_\texttt{b}(r) = \texttt{NULL}$ are set to $\texttt{STATE\_incoming}_\texttt{b}(r)$:

```
for all registers r
    if r ∈ HOT_current_b ∧ (STATE_incoming_b(r) = HOT ∨ STATE_incoming_b(r) = COLD)
        issue READ of r
        remove r from HOT_current_b
        if (STATE_b(r) = NULL)
            STATE_b(r) = GROUNDED;
    if (STATE_b(r) = NULL)
        STATE_b(r) = STATE_incoming_b(r);
```

This loop will terminate as each iteration examines one register and the number of registers is finite. After this code has executed, it can be asserted that: $\texttt{STATE}_\texttt{b}$ is correct; all necessary `READ`s to compensate for transitions between this block and scheduled successors have been issued; $\texttt{HOT\_current}_\texttt{b}$ is correct. As there are no more instructions that can alter $\texttt{HOT\_current}_\texttt{b}$, the following assignment is made:

$\texttt{HOT\_final}_\texttt{b} = \texttt{HOT\_current}_\texttt{b}$

Once this assignment is complete, it can be asserted that $\texttt{HOT\_final}_\texttt{b}$ is correct. Finally, SCHEDULED(b) is updated using:

SCHEDULED(b) = TRUE

And all of the invariants hold for the current block.

As this loop will be executed exactly once for each block $b \in \mathbf{B}$, and there are a finite number of blocks in $\mathbf{B}$, the loop will terminate. After each iteration of the loop

SCHEDULED(b) holds for one more block than it had on the previous iteration. Therefore, when the loop terminates the postcondition:

$b \in \mathbf{B} \rightarrow$ SCHEDULED(b) = TRUE

has been satisfied.

5.4.1 Correctness of the routine `Schedule()`

Recall that the required postconditions of this routine are:

- All original operations in **B** have been issued.
- All required `READS` have been inserted in **B**.
- $\texttt{HOT\_current}_b$ is correct up to this point.
- $\texttt{STATE}_b$ is correct up to this point.

Recall also, that the following preconditions have been shown to hold on entry to this routine:

- $b \in \mathbf{B} \wedge$ SCHEDULED(b) = FALSE
- $\texttt{HOT\_current}_b$ contains only registers that are in $\texttt{HOT\_final}_p$ of scheduled predecessors of b, and contains all such registers

This entire routine is in the form of a single loop denoted by:

for each operation O

As a block must have a finite number of operations, and each operation is dealt with in a single loop iteration, this loop will terminate. As the bottom of this loop is the instruction:

issue O

it can be trivially asserted that all original operations in b are issued.

Depending on the type of operation currently being handled, one of three clauses is executed. By showing that each clause properly handles one type of operation, it will be demonstrated that all operations will be properly handled.

The only item of concern for processing a `READ` is to ensure that following the `READ`, its *source* register is not in $\texttt{HOT\_current}_b$. In terms of $\texttt{STATE}_b$, if this is the first reference to the *source* register, its state is set to `GROUNDED`, and if this is the first reference to the *destination* register its state is set to `COLD`. This is all accomplished with the clause:

```
if STATE_b(source(O)) = NULL
    STATE_b(source(O)) = GROUNDED
if STATE_b(target(O)) = NULL
    STATE_b(target(O)) = COLD
if source(O) ∈ HOT_current_b
    remove source(O) from HOT_current_b
```

Following this clause and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a READ, all postconditions hold at the bottom of the loop after the first iteration.

When dealing with a WRITE, if the *target* register is in $\texttt{HOT\_current}_b$, it is necessary to insert a READ of this register. Following this insertion, the register is removed from $\texttt{HOT\_current}_b$. In either case, if this is the first reference to the *target* variable, its state is set to COLD. These manipulations are performed by:

```
if target(O) ∈ HOT_current_b {
    issue READ of target(O)
    remove target(O) from HOT_current_b
}
if STATE_b(target(O)) = NULL
    STATE_b(target(O)) = COLD
```

Following this clause and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a WRITE, all postconditions hold at the bottom of the loop after the first iteration.

When dealing with a LOAD, if the *target* of the LOAD is in $\texttt{HOT\_current}_b$, it is necessary to insert a READ of this register. The register is not removed from $\texttt{HOT\_current}_b$ however, as the *target* of the LOAD currently being processed must be in $\texttt{HOT}_b$ once the current instruction has been issued. If we are to issue this read of the *target* register, and this is the first reference to this register, its state is set to GROUNDED. If the *target* is not in $\texttt{HOT\_current}_b$, it is added to $\texttt{HOT\_current}_b$. If this is the first reference to this register, its state is set to HOT. This logic is encoded as:

```
if target(O) ∈ HOT_current_b {
    issue READ of target(O)
    if STATE_b(target(O)) = NULL
        STATE_b(target(O)) = GROUNDED
}
else {
    add target(O) to HOT_current_b
```

```
    if STATE_b(target(O)) = NULL
        STATE_b(target(O)) = HOT
}
```

Following this clause, and the issuing of the instruction at the bottom of the loop, it can be asserted that if the first operation is a `LOAD`, all postconditions hold at the bottom of the loop after the first iteration.

It can now be asserted that if the instruction is any of `READ, WRITE, LOAD`, following the issuing of the instruction at the bottom of the loop all postconditions hold after the first iteration. This can be generalized to the statement that all postconditions will hold at the bottom of any loop iteration, in particular the final loop iteration. Therefore, all postconditions hold at the end of this routine.

5.4.2 Correctness of the routine `JoinSuccessorStates()`

Recall that the requirement of this routine is that it has as its postcondition:

$\texttt{STATE\_incoming}_b$ is correct

To be able to discuss this, it is necessary that some description be provided as to what it means for this condition to hold. If $\texttt{STATE}_s(r)$ for a register is `HOT` for any scheduled successor s, then it is irrelevant what $\texttt{STATE}_s(r)$ for that register is for any other scheduled successor s. The register must be treated as if it is `HOT` for all successors. If $\texttt{STATE}_s(\texttt{r}) \neq$ `HOT` for all scheduled successors, but it is `COLD` for one or more successors, then it is treated as being `COLD` for all successors. If neither of these conditions are met, then the register is treated as being `GROUNDED` for all successors.

The preconditions for this routine are:

- $b \in \mathbf{B}$
- The information about successors to b is valid

The above description implies that the default setting for a register, unless it is overridden by a value from a scheduled successor, is `GROUNDED`. Therefore, the first operation in the routine is:

```
for all registers r
    STATE_b(r) = GROUNDED;
```

This loop will terminate as it iterates once per register and there are a finite number of registers. If there are no scheduled successors of b, then the routine terminates here, and the returned $\texttt{STATE}_b$ is `GROUNDED` for all registers. This is the correct return value for this circumstance. If there are scheduled successors, the loop denoted by:

for all scheduled successors of b

will be entered. This can be rewritten as:

$(\forall\, s \mid s \in \mathbf{B} \wedge \text{SCHEDULED}(s) = \text{TRUE} \wedge \text{SUCCESSOR}(b, s) = \text{TRUE})$

This loop will terminate as there must be a finite number of successors to a block. Within this loop, there is another loop which is denoted by:

for all registers r

This loop will also terminate as the number of registers is finite. The body of this loop is:

```
if (STATE_b(r) = HOT ∨ STATE_s(r) = HOT)
    STATE_b(r) = HOT;
else if (STATE_b(r) = COLD ∨ STATE_s(r) = COLD)
    STATE_b(r) = COLD;
```

The first check shown here ensures that if any scheduled successor has $\texttt{STATE}_\texttt{s}(r) = \texttt{HOT}$, then the set returned from this routine will have $\texttt{STATE}_\texttt{b}(r) = \texttt{HOT}$. Similarly, if none of the scheduled successors has $\texttt{STATE}_\texttt{s}(r) = \texttt{HOT}$, but one or more of them have $\texttt{STATE}_\texttt{s}(r) = \texttt{COLD}$, then the set returned from this routine will have $\texttt{STATE}_\texttt{b}(r) = \texttt{COLD}$. If none of the scheduled successors have either $\texttt{STATE}_\texttt{s}(r) = \texttt{HOT}$ or $\texttt{STATE}_\texttt{s}(r) = \texttt{COLD}$, then the set returned from this routine will have the default value of $\texttt{STATE}_\texttt{b}(r) = \texttt{GROUNDED}$. Upon assignment of the return value from this routine to $\texttt{STATE\_incoming}_\texttt{b}$, it can be asserted that $\texttt{STATE\_incoming}_\texttt{b}$ is correct.

5.5 Managing Transfer of Control

The preceding algorithm and proof do not address the issue of managing register state when a transfer of control, such as a function call or interrupt, occurs. It is assumed that interrupts are not an issue as they will either only use special hardware registers set aside for their use, or will first read any registers that they are going to write to preserve their initial values. The interrupt handler must take action to ensure that no registers which were not HOT on entry to the handler are HOT when control is returned to the user program.

The algorithm relies on all registers being in a known state upon entry to a function. In order to support separate compilation of source files and the use of libraries, it is necessary to ensure that no registers are HOT prior to issuing a function call. In a system where all source was required to be in a single input file, it would be possible to extend the interblock exchange of information to also handle interprocedural exchange. It is also possible to design a system that uses this interprocedural exchange of information by importing information from previously compiled images, but that is outside the scope of this paper.

6 Trace Scheduling

The previous sections have all dealt with a traditional compiler which works with basic block scheduling. The Multiflow compiler being used for the M-Machine work uses a trace scheduling algorithm [2]. In terms of the algorithm presented in this paper, the major difference between basic blocks and traces is that, unlike a basic block, a trace can have multiple entry and exit points. This section explains the changes that are necessary to the algorithm presented in Section 4 to accommodate trace scheduling; and discusses the impact of these changes on the demonstration of correctness.

6.1 The Trace Scheduling Algorithm

The motivation for trace scheduling is found in the problem of compiling for instruction-level parallelism (ILP). In general, it is difficult for a compiler to locate enough parallelism within a basic block to sustain utilization of multiple functional units. A compiler using the trace-scheduling algorithm seeks to overcome this obstacle by performing scheduling on a larger unit than a basic block. This larger unit is called a trace. A trace is allowed to span multiple basic blocks, and may contain conditional branches within it. Traces are not allowed to contain loop back edges. One important difference between a basic block and a trace is that a trace is allowed to have multiple exit and entry points.

An in-depth description of trace scheduling can be found in [2, 3].

6.2 Algorithm Modifications for Trace Scheduling

The first change that needs to be made to the algorithm is the structure with which state information is associated. For the basic block algorithm it makes sense to associate this information with each block. When dealing with trace-scheduling a more natural association is to be found with the edges in the control flow graph for the program. This provides a natural way of handling the multiple entry and exit points which can exist within a trace. One of the nice properties of these edges is that an edge will have only one entry point and one exit point.

At the start of processing for a trace, the algorithm gathers information from all edges which enter the trace at the top. After each instruction, it now becomes necessary to determine if there are any edges that either enter or leave the trace between the instruction that was just issued, and the next instruction to be issued. For all such edges, there is different handling depending on whether or not the edge has been scheduled. If there are scheduled edges joining the trace, then the HOT information from those traces is added into $HOT_{current}$. If the edge has not been scheduled, STATE is attached to the edge to be used when it is scheduled.

If there are unscheduled edges leaving the trace, $HOT_{current}$ is attached to the edge to be used when it is scheduled. If there are scheduled edges leaving the trace, a check

is made to determine if any synchronizing operations need to be inserted before the next instruction is emitted.

Based on these changes, the necessary changes to the algorithm can be formulated. As can be seen in Figure 10, the general outline of the algorithm is essentially the same; JoinSuccessorStates remains unchanged; but the algorithm for scheduling a single trace is different from that for a basic block.

```
while (not all traces scheduled)
    t = SelectTrace();
    HOT_initial_t = ∅;
    for all predecessors p of t
        if (predecessor p scheduled)
            HOT_initial_t = HOT_initial_t ∪ HOT_final_p;
    HOT_current_t = HOT_initial_t

    Schedule(t);
    STATE_incoming_t = JoinSuccessorStates(t);

    for all registers r
        if (r ∈ HOT_current_t AND (STATE_incoming_t(r) ≠ GROUNDED)
            issue read of r
            remove r from HOT_current_t
            if (STATE_t(r) = NULL)
                STATE_t(r) = GROUNDED;
        if (STATE_t(r) = NULL)
            STATE_t(r) = STATE_incoming_t(r);
    HOT_final_t = HOT
    Mark t as scheduled
```

Figure 10: General Outline of Trace Algorithm

The algorithm for scheduling the instructions within a trace is shown in Figure 11. This algorithm iterates over all of the instructions within a trace, ensuring that all operations are scheduled and all necessary READS are inserted. In addition, this algorithm determines the value of $\mathtt{STATE_t(r)}$ for all registers referenced in the block, and maintains the membership information for the set $\mathtt{HOT\_current_t}$. In addition, prior to issuing each instruction this algorithm determines if there are any splits from or joins to this trace entering at the current cycle. If there are such edges, the necessary manipulation of $\mathtt{STATE_t}$ and $\mathtt{HOT_t}$ are performed, and any READS necessitated by these edges are inserted.

```
for each cycle C
    for each edge e joining t at C
        if (edge e scheduled) {
            HOT_current_t = HOT_current_t ∪ HOT_final_e
        }
        else {
            Attach STATE_t to e
        }
    for each operation O {
        ⋮
        issue O
    }
    for each edge e splitting from t at C
        if (edge e scheduled) {
            for all registers r
                if (r ∈ HOT_current_t AND (STATE_incoming_e(r) ≠ GROUNDED)
                    issue read of r
        }
        else {
            Attach HOT_current_t to e
        }
```

Figure 11: Scheduling Algorithm for a Trace

6.3 Correctness Modifications for Trace Scheduling

Most of the changes to accommodate trace-scheduling clearly have little impact on the previously demonstrated correctness of the algorithm. The one area which bears some examination is the code that occurs at the beginning and end of each cycle of scheduling. These pieces of code mirror in many respects the code at the beginning and end of each basic block/trace.

At the beginning of each cycle, a check is made to determine if any edges join the current trace at this cycle. If edges do join at this cycle, it is necessary to either incorporate `HOT` information from the edge if it has been scheduled, or to attach `STATE` information to the edge if it hasn't been scheduled. In the first case, this is directly analogous to incorporating $\texttt{HOT}_{\texttt{final}}$ from all predecessors to a given trace into $\texttt{HOT}_{\texttt{initial}}$ prior to beginning scheduling of the trace. For any given instruction, it is necessary to have incorporated information about all preceding instructions which have already been scheduled. For the latter case, some clarification is needed. When `STATE` is attached to an edge this does not mean that the current values of `STATE` that have already been computed for this trace are copied into the edge. It means that a new $\texttt{STATE}_{\texttt{e}}$ is initialized, and all further computation of `STATE` that occurs in the process of scheduling the current trace must update not only $\texttt{STATE}_{\texttt{t}}$ for the trace, but all $\texttt{STATE}_{\texttt{e}}$ for unscheduled edges that have already joined the trace. This process will ensure that when the edge is scheduled it is presented with a clear picture of what will happen to registers following the transition from the edge into the adjoining trace.

At the end of each cycle, management of edges that split from the current trace are handled. If these edges have already been scheduled, a check is made to determine if any registers need to be `GROUNDED` prior to transitioning to the edge. Ideally, any necessary compensation code can be conditionalized, so that it is only executed if the branch to the edge is to be taken, and hidden in a branch shadow. If the adjoining edge has not been scheduled, then a copy of $\texttt{HOT}_{\texttt{current}}$ is placed on the edge. Relative to this edge, this is equivalent to placing $\texttt{HOT}_{\texttt{final}}$ into an adjoining trace. If there are multiple edges, and it is not possible to conditionalize any inserted `GROUNDing` operations, all of the scheduled edges should be handled first, to minimize the number of registers in the set `HOT` copied into the unscheduled edges.

7 Summary

This paper describes a compiler algorithm for preventing write-after-write hazards in the absence of hardware interlocks. While the algorithm was motivated by the design of the MIT M-Machine, it is applicable to any architecture that does not prevent this hazard.

The algorithm has been shown to be correct. Implementation of the algorithm within the Multiflow Compiler targeted for the M-Machine is currently under way. Future work will detail the implementation details of the algorithm; and evaluate the cost of managing this issue in software rather than hardware.

References

[1] Dally, W. J., et al., "M-Machine Architecture v1.0," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Concurrent VLSI Architecture Memo 58, February, 1994.

[2] Ellis, John R., 'Bulldog: A Compiler for VLIW Architectures', The MIT Press, Cambridge, MA, 1986.

[3] , Lowney, P.G., Freudenberger, Stefan M., et. al., The Multiflow Trace Scheduling Compiler', J. of Supercomputing, v.7(1-2), 1993.